链滴

Netty 4.x User Guide 个人翻译

作者：ZephyrJung

原文链接：https://ld246.com/article/1484814347289

来源网站：链滴

官网链接请戳：http://netty.io/wiki/user-guide-for-4.x.html，由ZephyrJung翻译

表示Google翻译现在真是异常强大，我甚至觉得以后再碰到英文文档都不用人翻了，全文粘过去好了

翻译也是一个学习的过程，大家共同努力吧~~有空不妨到黑客派社区看看，大牛坐镇，应有尽有~~

戳戳

Netty 4.x用户手册

前言

问题

如今，我们使用目标通用的应用或库来进行相互间的通讯。例如，我们常使用HTTP客户端库来检索自服务端的信息，并通过网络服务（web service）来进行一个远程过程调用。然而，一个通用目标协议或者它的实现有时候并不能尽善尽美。就好像我们不会用一个通用目标的HTTP服务器来交换大文件，电子邮件信息，以及近乎实时的信息如金融信息和多人游戏数据。这些都要求针对特定的目标实现高度优化的协议。例如，你可能想要实现一个HTTP服务器能够针对基于Ajax聊天应用，流媒体大型文件传输来优化。你甚至会想设计并实现一个全新的协议来为自己的需求精确的量身定做。另一不可避免的场景是处理历史遗留的专用协议来保证和老系统的互通性。这个场景的重点在于如何能为标应用在不牺牲稳定和性能的前提下快速的实现一个协议。

解决方案

Netty项目是一个提供异步时间驱动网络应用框架和快速开发可维护的高性能高扩展性服务端和客户协议工具集的成果。

换句话说，Netty是一个NIO客户端服务端框架，它使得快速而简单的开发像服务端客户端协议的网应用成为了可能。它它极大的简化并流线化了如TCP和UDP套接字服务器开发的网络编程。

“快速且简便”不意味着目标应用将容忍维护性和性能上的问题。Netty在吸取了大量协议实现（如FP，SMTP，HTTP以及各种二进制，基于文本的传统协议）的经验上进行了精心的设计。由此，Nett成功找到了一个无需折衷妥协而让开发、性能、稳定性和灵活性相互协调的方法。

一些用户可能已经找到了其他的生成有同样有点的网络应用框架，并想知道Netty与这些有什么不同答案是它所依赖的哲学。 Netty是设计来给你在API和实现上最佳体验的。它不是有迹可循的东西，你在阅读这片指南将察觉这一哲学会让你玩Netty的生活变得更好过。

开始

这篇指南围绕着Netty的核心结构举出了几个例子来让你迅速上手。当你看完这篇文章后将能够立刻出一个基于Netty的客户端和服务端。

如果你更喜欢自顶而下的学习方法，可以从第二篇开始，架构概览，然后返回到这里。

开始之前

本篇文章所介绍的示例程序所需要的最小运行需求只有两点：最新版本的Netty和JDK 1.6及以上。最版本的Netty可以在项目下载页找到。

当你读的时候，可能对本篇介绍的这些类有很多疑问。当你想要深入了解这些时，请查询API手册。

篇文章里的所有类都会关联到在线API手册以便于你阅读。同时，如果你发现了有不对的信息，语法误，排版错误或者你有提高这片文档的好主意，请务必果断联系Netty项目社区 。

写一个Discard服务器

在Netty的世界里，最简单的协议不是"Hello World!"，而是DISCARD。这个协议将丢弃任何接收到数据，不做任何响应。

实现DISCARD协议要做的事情就是忽略所有接收到的数据。让我们直接从处理实现开始，它负责处理etty产生的I/O事件。

```
package io.netty.example.discard;
import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHanlderAdapter;
/**
 * 处理服务端通道
 */
public class DiscardServerHandler extends ChannelInboundHanderAdapter{ //(1)
  @Override
  public void channelRead(ChannelHandlerContext ctx,Object msg){ //(2)
    //默默的丢弃接收到的数据
    ((ByteBuf) msg).release(); //(3)
  }
  @Override
  public void exceptionCaught(ChannelHandlerContext ctx,Throwable cause){ //(4)
    //出现异常时关闭链接
    cause.printStackTrace();
    ctx.close();
  }
}
```

1. DiscardServerHandler继承了 ChannelInboundHandlerAdapter，它是ChannelInboundHandler一个实现。ChannelInboundHandler提供了各种事件处理方法供你重写。目前，它只需要继承ChanelInboundHandlerAdapter而不用自己实现处理接口。

2. 我们重写了 channelRead()事件处理方法。这个方法将在接收到信息时被调用，无论何时从客户接收到新的数据。在这个例子中，接收的信息的数据类型是ByteBuf。

3. 为了实现 DISCARD协议，处理器需忽略接收到的信息。ByteBuf是一个引用计数对象，它需要通过elease()方法进行显示的释放。请牢记，释放一切传递给处理器的引用计数对象是处理器应尽的责任通常，channelRead()处理方法像如下这样实现：

```
@Override
public void channelRead(ChannelHandlerContext ctx,Object msg){
 try{
   //对msg做些什么
 } finally{
   ReferenceCountUtil.release(msg);
 }
}
```

4. 当Netty因为I/O错误或处理器实现在进行事件处理中抛出异常时，将会调用 exceptionCaught()法事件处理。虽然这个方法的实现会根据你对于异常情况下的处理而有所不同，但在大多数情况下，被获取的异常应该记录日志，关闭与之相关的通道。例如，你可能想要在关闭连接之前发送一个包含误码的响应信息。

目前为止一切正常。我们已经实现了DISCARD服务的一半。剩下的是写一个main()方法来调用DiscarServerHandler来启动服务。

```
package io.netty.example.discard;
import io.netty.bootstrap.ServerBootStrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelIntializer;
import io.netty.channel.ChannelOption;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
/**
 * 忽略一切进来的数据
 */
public class DiscardServer{
  private int port;
  public DiscardServer(int port){
    this.port=port;
  }
  public void run() throws Exception{
    EventLoopGroup bossGroup=new NioEventLoopGroup(); //(1)
    EventLoopGroup workerGroup=new NioEventLoopGroup();
    try{
      ServerBootstrap b=new ServerBootstrap(); //(2)
      b.group(bossGroup,workerGroup)
        .channel(NioServerSocketChannel.class) //(3)
        .childHanlder(new ChannelInitializer<SocketChannel>(){ //(4)
          @Override
          public void initChannel(SocketChannel ch) throws Exception{
            ch.pipeline().addLast(new DiscardServerHandler());
          }
        })
        .option(CHannelOption.SO_BACKLOG,128) //(5)
        .childOption(ChannelOption.SO_KEEPALIVE,true); //(6)
      //绑定并开始接收输入的连接
      ChannelFuture f=b.bind(port).sync(); //(7)
      //在服务器套接字关闭之前保持等待
      //在这个例子中，这不会发生，但你可以优雅的做这件事
      //关闭服务
      f.channel().closeFuture().sync();
    } finally{
      workerGroup.shutdownGracefully();
      bossGroup.shutdownGracefully();
    }
  }
  public static void main(String[] args) throws Exception{
    int port;
    if(args.length>0){
      port=Integer.parseInt(args[0]);
    }else{
      port=8080;
    }
    new DiscardServer(port).run();
```

```
  }
}
```

1. NioEventLoopGroup是一个处理I/O操作的多线程事件循环。Netty为不同类型的传输提供了多种样的EventLoopGroup 的实现。在这个例子中我们要实现一个服务端应用，因此将用到两个NioEventoopGroup。第一个，通常称为‘工头’，接受一个输入连接。第二个，通常称为‘工人’，当工头受连接并将接收到的连接分配给工人时，工人将处理接收到的连接的流量。有多少线程被使用以及如匹配到创建的通道(Channel)取决于EventLoopGroup 的实现以及可能甚至通过构造器进行配置。

2. ServerBootstrap是一个用来启动服务器的辅助类。你可以使用通道来直接启动服务。然而需要知的是，这是一个乏味的过程，在大多数情况下你并不需要做这些。

3. 在这里，我们指定使用 NioServerSocketChannel 类，该类用于实例化新的通道(Channel)以接输入连接。

4. 这里被指定的处理器通常由新接收的通道评定。 ChannelInitializer是一个特殊的处理器，目的帮助使用者配置一个新的通道。你很可能想要为新通道配置一个ChannelPipeline 添加一些处理器如DscardServerHandler来实现你的网络应用。当应用变得复杂时，你可能会在管道上添加更多的处理器最终将这个匿名类升级为一个顶级类。

5. 你也可以设置指派给通道实现的参数。我们在写一个TCP/IP服务，所以我们被允许设置套接字选项如cpNoDelay和keepAlive。请查阅接口文档 ChannelOption 以及相应的ChannelConfig实现，来对持的ChannelOption有个宏观的概念。

6. 你注意到 option()和childOption()了么？option()是为了接受输入连接的NioServerSocketChanel 。childOption()是为了父ServerChannel所接受的通道，在本例中是 NioServerSocketChannel 。

7. 我们已经整装待发了。剩下的就是绑定端口并启动服务。在这里我们绑定端口 8080到机器上所NIC（网络接口卡，network interface card）。你可以使用不同的绑定地址调用bind()方法多次。

祝贺你，你刚刚完成了基于Netty的第一个服务。

查看接收的数据

现在我们已经写了自己的第一个服务器，我们需要测试一下它是否真的能工作。最简单的方式就是使用elnet命令。例如，你可以在控制台命令行键入telnet localhost 8080，然后输入一些东西。

然而，我们能说这个服务器正常工作了么？我们无法确定，因为他是一个丢包服务。你不会得到任何应。为了证明它确实工作了，让我们来修改一下服务，打印它所接收到的东西。

我们已经知道channelRead()方法会在任意数据接收是激活。让我们在DiscardServerHandler的channelRead()里放入一些代码：

```
@Override
public void channelRead(ChannelHandlerContext ctx,Object msg){
  ByteBuf in = (ByteBuf) msg;
  try{
    while(in.isReadable()){ //(1)
      System.out.print((char) in.readByte());
      System.out.flush();
    }
  } finally{
    ReferenceCountUtil.release(msg); //(2)
  }
}
```

1. 这个效率低下的循环其实可以简化为： System.out.pritnln(in.toString(io.netty.util.CharsetUtil

US_ASCII))

2. 这里可以做 in.release()，作为附加选项。

如果你再度运行telnet命令，你会发现服务器会打印出接收到的内容。

丢包服务器的全部代码位于发布包中的io.netty.example.discard。

编写一个响应服务器

到目前为止，我们一直在消费数据而不做任何响应。然而，一个服务器通常需要对一个请求作出响应现在让我们来学习一下如何通过实现一个 ECHO协议来向客户端响应信息，返回任何接收到的数据。

与先前的丢包服务器唯一不同之处在于，它将接收到的数据返回，而不是仅仅在控制台打印输出。因，只需要修改一下channelRead()方法就足够了：

```
@Override
public void channelRead(ChannelHandlerContext ctx,Object msg){
  ctx.write(msg); //(1)
  ctx.flush(); //(2)
}
```

1. 一个 ChannelHandlerContext对象提供了多种操作，使得你能够出发不同的I/O事件和操作。在里，我们触发write(Object)来逐字输出接收到的信息。请注意我们没有像在DISCARD例子中那样释接收的信息。因为在写出到线上时Netty将为你释放它。

2. ctx.write(Object)并没有将信息写出到线上。它在内部是缓冲起来，而后通过ctx.flush()来冲出。以通过调用ctx.writeAndFlush(msg)来简化。

当你再次运行telnet命令，你将看到服务器将返回一切你发送给它的内容。

响应服务器的所有代码位于发布的 io.netty.example.echo 包内。

编写一个时间服务器

这一节要实现的协议是TIME。这与前面的例子不同。它发送了一个包含32位整数的信息，且不接收何请求，并在信息发送完成后关闭连接。在这个例子里，你将学会如何组织并发送一个信息，然后在成时关闭连接。

由于我们打算忽略任何接收的数据，并且在连接创建时立刻发送一个信息，这次我们不能使用channeRead()方法。取而代之，我们需要重写channelActive()方法。实现如下：

```
package io.netty.example.time;
public class TimeServerHandler extends ChannelInboundHandlerAdapter{
  @Override
  public void channelActive(final ChannelHandlerContext ctx){ //(1)
    final ByteBuf time = ctx.alloc().buffer(4); //(2)
    time.writeInt((int)(System.currentTimeMillis()/1000L+2208988800L));
    final ChannelFuture f=ctx.writeAndFlush(time); //(3)
    f.addListener(new ChannelFutureListener(){
      @Override
      public void operationComplete(ChannelFuture future){
        assert f==future;
        ctx.close();
```

```
      }
    }); //(4)
  }
  @Override
  public void exceptionCaught(ChannelHandlerContext ctx,Throwable cause){
    cause.printStackTrace();
    ctx.close();
  }
}
```

1. 不言而喻， channelActive()方法将在连接创建时被激活，准备产生流量。让我们在这个方法中入一个32位整数来代表当前时间。

2. 为了发送新的信息，我们需要分配一个新的缓冲区来保存。32位整数需要一个容量为4个字节的 ByteBuf，通过ChannelHandlerContext.alloc()获取当前的ByteBufAllocator并分配一个新的缓冲区。

3. 像往常一样，我们来写信息的构造。

但是请等一下，flip在哪儿？难道在NIO中，发送信息之前不应该调用java.nio.ByteBuffer.flip()么？ByeBuf并不包含这个方法，因为它有两个指针。一个指向读操作，另一个是写操作。当你向ByteBuf写么东西的时候，写指针地址索引将会增长，读指针不变。读写地址索引分别代表了信息的开始与结束。

相反，如果没有flip方法，NIO缓冲区并没有提供一个简便的方式来辨别信息内容的开始和结束。如你忘记flip缓冲区将会陷入麻烦，因为空或者错误的数据将被发送。这样的错误不会在Netty中发生，为对于不同的操作类型，我们有不同的指针。当你习惯与此你会发现这让你的日子更好过了——没有flp out的日子!

另一个需要知道的点是ChannelHandlerContext.write() (以及writeAndFlush()) 方法返回一个 ChanelFuture。一个 ChannelFuture代表一个尚未发生的I/O操作。这意味着，任何请求操作可能都还没执行，因为Netty中所有操作都是异步的。例如，下面这段代码可能会在信息发送前就关闭了连接：

```
Channel ch=...;
ch.writeAndFlush(message);
ch.close();
```

因此，你需要在write()返回的ChannelFuture完成以后调用close()方法。它将会在写操作完成后通知听者。请注意，close()也可能不会立刻关闭连接，并且它返回一个ChannelFuture。

4. 那当请求完成后，我们如何获得通知呢？很简单，只需要在返回的 ChannelFuture上添加一个 ChnnelFutureListener。这里，我们创建了一个新的异步 ChannelFutureListener来在操作结束时关闭道。

另一种选择，是使用预定义的监听器来简化代码：

```
f.addListener(ChannelFutureListener.CLOSE);
```

为了测试我们的时间服务器像预期的那样工作，可以使用UNIX的rdate命令：

```
$ rdate -o <port> -p <host>
```

<port>是你在main()方法中定义的端口号，<host>一般是localhost。

编写一个时间客户端

与DISCARD和ECHO服务器不同，我们需要为TIME协议准备一个客户端，因为人无法手工将32位二制数据转换为日历上的一个日期。在这一节，我们讨论如何确保服务端正确的工作，并学习如何用Nety写一个客户端。

Netty关于服务端和客户端最大和唯一的区别在于使用了不同的Bootstrap 和Channel实现。请看一下面的代码：

```
package io.netty.example.time;
public class TimeClient{
  public static void main(String[] args) throws Exception{
    String host=args[0];
    int port=Integer.parseInt(args[1]);
    EventLoopGroup workerGroup=new NioEventLoopGroup();
    try{
      Bootstrap b=new Bootstrap(); //(1)
      b.group(workerGroup); //(2)
      b.channel(NioSocketChannel.class); //(3)
      b.option(ChannelOption.SO_KEEPALIVE,true); //(4)
      b.handler(new ChannelInitializer<SocketChannel>(){
        @Override
        public void initChannel(SocketChannel ch) throws Exception{
          ch.pipeline().addLast(new TimeClientHandler());
        }
      });
      //启动客户端
      ChannelFuture f=b.connect(host,port).sync(); //(5)
      //在连接关闭之前保持等待
      f.channel().closeFuture().sync();
    }finally{
      workerGroup.shutdownGracefully();
    }
  }
}
```

1. Bootstrap与 ServerBootstrap 类似，只不过它用于非服务端通道，如客户端或无连接通道。
2. 如果你只指定了一个 EventLoopGroup，它将既做工头也做工人。尽管工头并不用于客户端。
3. NioSocketChannel 用来创建一个客户端的通道，而不是 NioServerSocketChannel。
4. 注意这里我们没有使用 childOption()，因为客户端的 SocketChannel 没有父类。
5. 我们应当调用 connect()方法而非bind()方法。

如你所见，这与服务端代码不尽相同。那么ChannelHandler 实现又怎样呢？他应该从服务器接收32整数，并翻译成人工可读格式，打印翻译的时间，关闭连接：

```
package io.netty.example.time;
import java.util.Date;
public class TimeClientHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ByteBuf m = (ByteBuf) msg; // (1)
        try {
            long currentTimeMillis = (m.readUnsignedInt() - 2208988800L) * 1000L;
            System.out.println(new Date(currentTimeMillis));
            ctx.close();
        } finally {
            m.release();
        }
```

```
    }
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        cause.printStackTrace();
        ctx.close();
    }
}
```

1. 在TCP/IP中，Netty从对等端读取数据到一个 ByteBuf。

这看起来很简单，与服务端的示例没有什么大不同。然而，这个处理器有时候会拒绝工作并发出一个IndexOutOfBoundsExcepiton。在下一节我们来讨论为什么这个会发生。

处理基于流的传输

关于套接字缓冲区的一个小注意事项

在基于流传输如TCP/IP中，接收的数据存储在一个套接字缓冲区中。不幸的是，这个流传输缓冲区并是一个包的队列，而是一个字节队列。意思是说，即便你发送了两个独立包的信息，操作系统并不会它们当做两条信息，而是当做一堆字节。因此无法保证你所读的内容就是远程另一端所写的内容。例，假定一个操作系统的TCP/IP栈接收到了三个包：

ABC DEF GHI

由于基于流协议的一般属性，在你的应用里有很大概率读取为如下的片段形式：

AB CDEFG H I

因此，在接收的部分，不论是服务端还是客户端，都应该将接收到的数据分割成一个或者多个有意义帧，以便能够被应用逻辑方便的理解。如上面这个例子，接收的数据应该封装成为下面这样的帧：

ABC DEF GHI

第一个方案

现在，让我们返回TIME客户端这个例子。这里有同样的问题。一个32位整数是一个很小量的数据，且它不可能经常被碎片化。然而，问题是它可以被碎片化，并且随着流量的提高，碎片化的可能性也提高。

最简单的办法是创建一个内部的累积缓冲区，等待所有4字节被放进来。如下是一个修改后的TimeClietHandler，解决了这个问题：

```
package io.netty.example.time;
import java.util.Date;
public class TimeClientHandler extends ChannelInboundHandlerAdapter {
    private ByteBuf buf;
    @Override
    public void handlerAdded(ChannelHandlerContext ctx) {
        buf = ctx.alloc().buffer(4); // (1)
    }
    @Override
    public void handlerRemoved(ChannelHandlerContext ctx) {
        buf.release(); // (1)
```

```
        buf = null;
    }
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ByteBuf m = (ByteBuf) msg;
        buf.writeBytes(m); // (2)
        m.release();
        if (buf.readableBytes() >= 4) { // (3)
            long currentTimeMillis = (buf.readUnsignedInt() - 2208988800L) * 1000L;
            System.out.println(new Date(currentTimeMillis));
            ctx.close();
        }
    }
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        cause.printStackTrace();
        ctx.close();
    }
}
```

1. 一个 ChannelHandler 有两个生命周期监听器方法：handlerAdded()和handlerRemoved()。可以执行任意一个初始化（反初始化）任务，只要它没有长时间阻塞。

2. 首先，所有接收的数据都应该积累到 buf中。

3. 然后，这个处理器必须检查 buf是否包含了足够的数据，这个例子里是4个字节，然后再进行真的业务逻辑。当有更多数据到达时，Netty将会再次调用channelRead()方法，最终将累积所有4个字。

第二个方案

虽然第一个方案解决了TIME客户端的问题，但是修改后的处理器看起来不甚简洁。想象一个更复杂协议，融合了如变量长度的多个字段。你的 ChannelInboundHandler 实现将很快变得无法维护。

你可能已经注意到了，你可以在ChannelPipeline上添加不止一个 ChannelHandler ，因此，你可以单个的 ChannelHandler 划分成多个模块，来降低你应用的复杂度。例如，你可以将TimeClientHandler划分成两个处理器：

- TimeDecoder用来处理碎片问题，以及
- 初始化 TimeClientHandler的简单版本

幸运的是，Netty提供了一个可扩展的类来帮你写出第一个东西：

```
package io.netty.example.time;
public class TimeDecoder extends ByteToMessageDecoder { // (1)
    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) { // (2)
        if (in.readableBytes() < 4) {
            return; // (3)
        }
        out.add(in.readBytes(4)); // (4)
    }
}
```

1. ByteToMessageDecoder 是 ChannelInboundHandler 的一个实现，它使得处理碎片问题变得更易了。

2. ByteToMessageDecoder 调用了decode()方法，这个方法内部维护了一个累积缓冲区来接收任何来的数据。

3. decode()能够决定在没有累积缓冲区足够的数据时不向out添加内容。 ByteToMessageDecoder 会在更多数据接收时再次调用decode()。

4. 如果 decode()向out添加了一个对象，这意味着解码器成功的解码了一个信息。 ByteToMessagDecoder 将会放弃读取部分累计缓冲区。请记住你没有必要解码多个信息。ByteToMessageDecoder 将会持续调用decode()方法直到没有可向out添加的内容为止。

现在我们有另一个处理器添加到 ChannelPipeline 中，我们需要修改TimeClient中的 ChannelInitializer 实现：

```
b.handler(new ChannelInitializer<SocketChannel>() {
    @Override
    public void initChannel(SocketChannel ch) throws Exception {
        ch.pipeline().addLast(new TimeDecoder(), new TimeClientHandler());
    }
});
```

如果你乐于冒险，可以尝试使用 ReplayingDecoder 让解码器进一步简化。当然，你需要查询API手来获取更多信息。

```
public class TimeDecoder extends ReplayingDecoder<Void> {
    @Override
    protected void decode(
            ChannelHandlerContext ctx, ByteBuf in, List<Object> out) {
        out.add(in.readBytes(4));
    }
}
```

另外，Netty提供了开箱即用的解码器，可以让你很容易的实现大多数协议，帮你避免以单个无法维的处理器实现告终。请查看下面的包中的例子来获取更多细节：

- io.netty.example.factorial 二进制协议
- io.netty.example.telnet 文本行协议

用POJO说话而非ByteBuf

目前我们所看到的代码都是用 ByteBuf 作为协议信息的主要数据结构。在这一节，我们将升级TIME议的客户端和服务端例子，使用POJO而不是 ByteBuf。

在 ChannelHandler 中使用POJO的有点显而易见。你的处理器将更容易维护，即从ByteBuf抽取的息能够被分离的代码重用。在TIME客户端和服务端的例子中，我们只读取了32位整数，而直接使用ByeBuf并不是主要问题。然而，当你实现一个现实世界中的协议时，你会发现做分离是很有必要的。

首先，让我们定义一个新的类型，名为UnixTime。

```
package io.netty.example.time;
import java.util.Date;
public class UnixTime {
    private final long value;
```

```
    public UnixTime() {
        this(System.currentTimeMillis() / 1000L + 2208988800L);
    }
    public UnixTime(long value) {
        this.value = value;
    }
    public long value() {
        return value;
    }
    @Override
    public String toString() {
        return new Date((value() - 2208988800L) * 1000L).toString();
    }
}
```

现在我们可以修改TimeDecoder来产生一个UnixTime而非 ByteBuf。

```
@Override
protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) {
    if (in.readableBytes() < 4) {
        return;
    }
    out.add(new UnixTime(in.readUnsignedInt()));
}
```

使用了新的解码器，TimeClientHandler将不再使用 ByteBuf ：

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    UnixTime m = (UnixTime) msg;
    System.out.println(m);
    ctx.close();
}
```

更简单优雅了，对么？同样的技术将应用在服务端。这次让我们更新一下TimeServerHandler：

```
@Override
public void channelActive(ChannelHandlerContext ctx) {
    ChannelFuture f = ctx.writeAndFlush(new UnixTime());
    f.addListener(ChannelFutureListener.CLOSE);
}
```

现在，唯一缺少的部分就是编码器了，它是 ChannelOutboundHandler 的一个实现，将UnixTime 译回一个 ByteBuf 。这比写一个解码器容易得多，因为在对信息编码时不需要处理数据包分片和组装。

```
package io.netty.example.time;
public class TimeEncoder extends ChannelOutboundHandlerAdapter {
    @Override
    public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) {
        UnixTime m = (UnixTime) msg;
        ByteBuf encoded = ctx.alloc().buffer(4);
        encoded.writeInt((int)m.value());
        ctx.write(encoded, promise); // (1)
    }
}
```

1. 这一行有几个重要的地方：

首先，我们传递了一个原生态 ChannelPromise ，当编码后的数据实际向线上写出时，Netty标记它成功或者失败。

其次，我们不调用ctx.flush()。这里有一个处理器方法void flush(ChannelHandlerContext ctx)，它目的就是重写flush()操作。

为了更进一步简化，你可以使用 MessageToByteEncoder：

```
public class TimeEncoder extends MessageToByteEncoder<UnixTime> {
    @Override
    protected void encode(ChannelHandlerContext ctx, UnixTime msg, ByteBuf out) {
        out.writeInt((int)msg.value());
    }
}
```

最后一个剩下的任务是将TimeEncoder在TimeServerHandler之前插入到服务端的 ChannelPipeline ，这作为一个练习任务略过。

关闭应用

关闭一个Netty应用通常和使用shutdownGracefully()关闭所有你创建的EventLoopGroup一样简单它返回一个 Future 来告诉你 EventLoopGroup 何时完全终止，以及所有属于这个组的Channel都已关闭。

总结

在本篇文章中，我们使用一个关于如何基于Netty写一个完全工作的网络应用的展示进行了一次快速Netty之旅。

即将到来的篇章中将包含关于Netty的更多章节，我们鼓励你复读 io.netty.example 包下的Netty示。

同时请知道 the community 永远在等待你的问题和反馈来帮助提高Netty及其文档。